WARATEK

WHITEPAPER

An Introduction to Software-Defined Runtime Application Self-Protection (RASP)

This whitepaper aims to provide a clear starting point to Software-Defined Security knowledge. We make a conscious effort to describe RASP objectively and optimistically for what it can be.

A Shift Has Happened

We live in a world where 84% of software exploits happen at the application layer. Yet we continue to rely on vintage security techniques at the network layer to protect enterprise applications and the millions of users that use them.

Whether your organization uses a WAF, RASP, or a combination of SAST, DAST, or IAST, the only reliable approach to address these vulnerabilities is to patch the codebase.

Still, we make assumptions about risk in the form of heuristics that require a significant amount of manual investigation. Today enterprises deploy code multiple times a day, and Security teams must keep pace with each deployment where each code change can introduce new and previously patched vulnerabilities.

Three factors make this increased speed unsustainable for Security teams:

1. Fixing vulnerabilities is manual
2. Existing tooling adds noise rather than value
3. Code changes lead to vulnerability regressions

This never ending effort keeps them running on a metaphoric treadmill where they move fast but go nowhere.

The reality is simple - the current approaches are not sustainable for security professionals. If you need proof that the status quo is no longer adequate, look no further:

- A record-breaking number of 3,205 data breaches occurred in 2023.
- Gartner predicts that 99% of successful cyberattacks will continue to result from known but unpatched software vulnerabilities.
- NIST adds a new software vulnerability to the National Vulnerability Database every 30 minutes on average.

A 2022 report from Palo Alto Networks stresses how hackers start scanning for vulnerabilities within just 15 minutes once a new CVE is published.

With hackers becoming more dangerous than ever in recent years, it can take them mere minutes to find a weak point in their target's applications.

In contrast, the average time to fix these critical software vulnerabilities sits at an all-time high of 205 days.

If traditional cybersecurity approaches aren't decreasing the frequency of attacks or the time it takes to patch the vulnerabilities, how do we, as a community, improve our approach?

Introducing Software-Defined RASP

Software-Defined RASP is the practice of leveraging machine-readable definition files that use high-level descriptive coding language to apply immutable and continuous security behavior unlike that found in RASPs that require sampling.

This approach drastically reduces reliance on human intervention and grants security teams autonomy while allowing engineers to focus on development rather than vulnerability remediation ensuring both teams maintain their KPIs.

While there are many resources available to learn about RASPs, most RASPs fall short of fully aligning with their theoretical definition due to several practical and technical limitations:

- **Performance Overhead:** Many RASP solutions introduce significant latency, forcing them to rely on sampling rather than continuous real-time protection. This leads to delayed detection and response times, which is more akin to traditional security mechanisms like Web Application Firewalls (WAFs).
- **Scalability Issues:** High-volume and dynamic workloads make real-time analysis impractical, leading to scalability challenges. Sampling provides a way to monitor security without overwhelming the system, but it sacrifices real-time protection.
- **Integration Limitations:** RASP solutions often function more as external modules, missing the deep integration needed for context-aware security. This lack of integration can lead to less accurate threat detection as the RASP might not fully understand the context of the application's operations.
- **Resource Constraints:** Real-time comprehensive security analysis demands substantial CPU and memory resources, which can be impractical for many production environments. This often forces vendors to use less resource-intensive methods like sampling, which compromises the real-time nature of protection.

Waratek's Approach

Waratek takes a new approach to providing runtime protection for applications today. Central to Waratek's approach is the utilization of non-heuristic detection techniques for code-injection attacks based on two features: data tainting & syntax analysis.

Waratek deliberately chose not to use existing heuristics-based techniques for detecting code-injection attacks as used by Web Application Firewalls (WAFs), such as pattern matching, regular expressions, exploit signatures, blacklists, or whitelists. This decision avoids the inaccuracies, false-positives, continuous tuning, and performance degradation that plague heuristic-based approaches.

Data Tainting

Data tainting (also known as taint checking) marks as "untrusted" all user-input data to a Java app (like HTTP request parameters). This aids in distinguishing between developer-written code and user-inputted data.

Syntax Analysis

Once data is tainted, Waratek performs syntax analysis to identify if the user-input data is a code-injection exploit. For example, by intercepting SQL statements before they reach the database, Waratek can detect SQL injection attacks according to the formal grammar of the SQL dialect of the application's database.

For example, consider the illustration of a SQL statement composed by a programmer with user-inputted data (i.e., param), and how the Oracle PL/SQL database might represent the abstract syntax tree graph for this statement. When user-input data is well-formed and comprises only a single value, the statement is 'safe' and is not a code-injection attack. However, if manipulated by malicious user input to introduce an SQL injection exploit, the abstract syntax tree graph shows more nodes than previously, a condition described as a syntax overflow.

By performing attack analysis in this manner, Waratek avoids all the false-positive risks of heuristic-based approaches, resulting in the following benefits.

1. Immutability

Immutability is a cornerstone of effective security, ensuring that the protective measures you define remain consistently active, preventing vulnerability regressions with each deployment and user input. Security must be continuous and integrated within the runtime environment to achieve true immutability, rather than being a one-time operation in the CI/CD pipeline.

For security to be continuous, it must be integrated within the runtime or virtual machine environment. Static analysis tools, while helpful, only marginally improve the situation by identifying code that *might* needs fixing. This still leaves teams with the manual and labor-intensive task of patching vulnerabilities.

The key to improving security scalability is to automate the process of patching code. Compilers have operated on this principle for decades, enhancing performance through automation. Applying this philosophy to security allows for the automatic correction of known vulnerabilities and the insertion of security rules to block unknown vulnerabilities (zero days).

Automating these processes eliminates false positives and negatives, and reduces the need for expensive hardware, making security efforts more efficient. This decreased friction enables security teams to scale effectively with modern software development practices for the first time.

2. Scalability

Cost and human capital are significant barriers to the scalability of current security solutions. Traditional Web Application Firewalls (WAFs) and Runtime Application Self-Protection (RASP) platforms function as massive data pipelines, ingesting, analyzing, and making heuristic-based assumptions on HTTP payloads to assess risk. This reliance on HTTP payloads, which are lagging indicators, leads to a high incidence of false positives and negatives, exacerbating the scalability issues of existing security solutions.

To achieve true scalability in security, automation is essential. Just as compilers have automated performance optimization for decades, the same philosophy can be applied to security. By automating the process of patching known vulnerabilities and inserting security rules to block unknown vulnerabilities (zero days), we can eliminate false positives and negatives and reduce the need for expensive hardware.

This approach significantly reduces friction, enabling security teams to scale their efforts in line with modern software development practices for the first time.

3. Performance

While theoretical infallible security solutions might seem ideal, they must also meet real-world business demands. Consequently, many security solutions recommend running in sample mode, protecting only a subset of requests to minimize performance impact.

However, this trade-off is unnecessary when your RASP is deeply integrated with applications within the runtime. By adopting principles from compiler design, Software-Defined RASP implementations can achieve comprehensive protection with minimal performance overhead.

Software-Defined RASP solutions average less than a % performance impact at scale. This allows companies to achieve robust security without compromising on performance, ensuring they can stay both protected and competitive.

What Does this Look Like in Practice?

Software-Defined RASP does not require direct interaction with the application code or prior knowledge of the application.

When executing a method on your application for the first time and attempting to exploit a vulnerability, the JVM or CLR replaces the execution in milliseconds by performing a checksum check without downtime, source code changes, or tuning.

On any additional calls to that same method, only the protected version of your code is available, resulting in even faster execution.

Compared to the complex nature of a WAF or traditional RASP and the overhead of managing large numbers of rules, the business benefits are clear: unprecedented accuracy, low maintenance, virtually zero performance overhead, and no disruption of service once deployed.

Declarative and Immutable Rules

In declarative approaches, you specify the desired final state of the security you want to apply without dictating how to get it.

This behavior is ideal for Common Weakness Enumerations (CWEs), which act as dictionaries of vulnerabilities and refer to software weaknesses rather than specific instances of vulnerabilities.

Declarative programming minimizes the factors that could affect the behavior of a piece of code to ensure that only a function's input should affect the output, not what's happening elsewhere in the program.

This approach enables Software-Defined RASP's immutability and makes it possible to prevent vulnerability regressions.

For example, if SQLi vulnerabilities are rampant in your applications, it's possible to declaratively tell your applications that you never want to see another SQLi again.

From the time you deploy your SQLi rule, there's no code that developers can add to the codebase that will override your rule.

Due to the nature of these rules, they inherently remediate zero-days in some instances. This inherent protection is the case for CVE-2022-42889, in which the process forking rule is more than sufficient for payloads supplied as Javascript code, as seen below:

```
app("nashorn CVE-2022-42889"):
    requires(version: ARMR/2.2)
    process("Deny any process execution"):
        execute("*")
        protect(message: "", severity: 7)
    endprocess
endapp
```

Software-Defined RASP also provide imperative rules for more involved vulnerabilities like Log4shell (CVE-2021-44228), where conditionality and context are needed.

Imperative and Instant Time-to-Remediate

In the imperative approach, Software-Defined RASP allows you to prepare automation scripts that apply security one step at a time. This method requires extensive domain knowledge but offers greater control over vulnerability remediation, ideal for making specific adjustments or optimizations.

For example, with Log4shell, removing the JNDI Lookup class as recommended can break some applications. Instead, imperative rules let you specify conditions where the class functions and set constraints on its outputs.

Below is an example imperative rule for CVE-2021-44228:

```
app("APACHE LOG4J - CVE-2021-44228 - v1.2, b2"):
    requires(version: ARMR/2.2)

    patch("CVE-2021-44228 :01"):
    function("org/apache/logging/log4j/core/net/JndiManager.
lookup(Ljava/lang/String;)Ljava/lang/Object;",
        checksums: ["da55340ac1",
                    "02c6120d62",
                    "b04cf027e3",
                    "d3ad3c6d00",
                    "bb0462e72d"])
        entry()

        code(language: java):
            public void patch(JavaFrame frame) {
                String payload = frame.
loadStringVariable(1);
                log("Forcing JndiManager.lookup() to return
'null' due to CVE-2021-44228", payload);
                frame.returnObject(null);
            }

            private static void log(String msg, String
payload) {
                ArmrEvent event = ArmrEvent.load("ALERT",
"HIGH");
                event.addExtension("msg", msg);
                event.addExtension("payload", payload);
                event.commit();
            }
        endcode
    endpatch
endapp
```

Summary

Enterprises like yours use Waratek to provision, apply, and orchestrate application security for all their applications where it's most impactful - the live binary.

Our products are standardized across the ecosystem, used by security practitioners and leaders worldwide, and trusted by enterprises everywhere.

Prior to the introduction of Software-Defined RASP, it's never been economically feasible to secure every application and significantly lower your risk profile.

However, Software-Defined RASP enables you to fix the executing code rather than make assumptions on a lagging indicator like network data, and the following happens:

- False positives and negatives are non-existent
- The human capital needed to secure applications decreases
- The economics of securing every app becomes possible

For more information on Waratek visit Waratek.com.

> **Utilizing Software-Defined RASP enables organizations to scale with modern software development by codifying security and policy into development processes and workflows.**
>
> – Melinda Marks, Senior Analyst, ESG

How it Works

Waratek offers a novel approach to Application Security through it's Secure Execution agent. As a plugin to the Java runtime, Waratek provides runtime protection without requiring code changes utilizing:

- **Data Tainting:** Marks all user inputs as "untrusted" in real-time, distinguishing between developer-written code and user-supplied data
- **Syntax Analysis:** Performs lightweight analysis of user input according to the structured grammar of the input syntax, allowing Waratek to detect code injection attacks by identifying syntax overflows where malicious input creates unexpected nodes in the syntax tree

Unlike heuristic-based security solutions, Waratek's non-heuristic approach aims to eliminate false positives while providing effective protection against zero-day threats and code injection attacks.

Assuming an app lets users upload and display profile pictures. The process involves uploading a file via an endpoint, checking if the file has an allowed extension, saving the file to the server if allowed, and updating the user's profile and serving the new URL.

Waratek tracks the flow of data, logging changes, allowing it to identify and fix vulnerabilities at any point in the app's lifetime, by correcting the flawed code.

Technical Specs

Feature	Notes
Agent Size	3MB
CPU Utilization	< 2%
Memory utilization	25MB
Network Utilization	Negligible at scale